NASA-TM-112300

7N-61-TM

A Standard C Port of the NAS Kernels Benchmark Program

I. E. Stockdale[1]

Report RND-93-008 May 1993

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035

ARC 275 (Rev Feb 81)

A Standard C Port of the NAS Kernels Benchmark Program

I. E. Stockdale[1]

Report RND-93-008 May 1993

NAS Systems Development Branch
NAS Systems Division
NASA Ames Research Center
Mail Stop 258-6
Moffett Field, CA 94035-1000

Abstract

FORTRAN is widely viewed as the programming language of choice for scientific and numerical computations. However, limitations of the language lead some to argue that it should be replaced by another language which better lends itself to the production of structured and maintainable code. We provide quantitative data relevant to this argument by reporting on a port of the NAS Kernels benchmark program from FORTRAN 77 to ANSI/ISO Standard C.

Single-processor performance results for the CRAY Y-MP, CRAY-2, Convex 3240 and SGI 4D/25 are reported for C and FORTRAN versions of the code. Strengths and weaknesses of C *vis a vis* FORTRAN are evaluated. While the structured programming features of C are quite desirable, FORTRAN arrays provide the programmer with much more flexibility and aid the compiler in producing optimized code.

[1] Computer Sciences Corporation, NASA Ames Research Center, Moffett Field, CA 94035-1000

1. Introduction

FORTRAN has long been the *de facto* standard programming language in the scientific and numerical analysis communities. In recent years, C has become widely used beyond its original role as a systems programming language, and is sometimes suggested as a suitable language for scientific programming. Standard ANSI/ISO C [1] is becoming widely available, placing the language on the same footing as FORTRAN 77 [2]. Fortran 90 [3], recently adopted by ANSI and ISO, is to date only available from third party software houses and cannot yet be compared to either FORTRAN 77 or Standard C.

Several C language features make it an attractive choice for numerical computing. *Structs* provide a natural way to combine related data in a single structure. Pointers and the ability to dynamically allocate memory allow the programmer to handle varying amounts of data in a reasonable way. Include files and scoping rules allow better control of access to data by different subroutines.

In addition to the inevitable hurdles facing adoption of a new language, there are several potential problems which must be addressed when considering a change from FORTRAN to C. Pointers, cited as an improvement above, can hurt performance by making it more difficult for the compiler to analyze and optimize code. C implements multi-dimensional arrays in a less flexible way than FORTRAN, leading to questions about its effectiveness in large scale numerical computations.

In this paper a port of an existing benchmark code is discussed, with particular emphasis on testing the weak points of C *vis a vis* FORTRAN. The code selected for this project was the NAS Kernels benchmark program [4].

1.1 Description of kernels

The NAS Kernels Benchmark Program is a collection of FORTRAN 77 subroutines which characterize vectorized Computational Fluid Dynamics (CFD) codes running on the NAS high speed processors. The kernels represent algorithms which are used inside application programs, but are not complete applications. Each kernel is called by its own driver subroutine, which sets up input data arrays, performs timing, and calculates the fractional deviation of a single output array element from a pre-determined reference value.

Table 1 contains the number of floating point operations (flops) in each kernel. These numbers are computed by the benchmark code for each run. The flop counts are identical to those produced by the

FORTRAN version of the code and do not vary from computer to computer, providing a meaningful basis for comparison between executables. Performance results in MFLOPS (Millions of Floating Point Operations per Second) are calculated using the numbers in Table 1. The sum of the fractional errors for all kernels may not exceed 5 x 10^{-10}. Performance results on the FORTRAN 77 NAS Kernels have been reported previously [5]. The code has been ported to an early version of Fortran 90 [6].

Table 1: Flop counts for the NAS Kernels

Kernel	Number of flops
mxm	4.1943×10^8
fft	4.9807×10^8
Cholesky	2.2103×10^8
btrix	3.2197×10^8
gmtry	2.2650×10^8
emit	2.2604×10^8
vpenta	2.5943×10^8

1.1 Description of tested machines

The performance results described in this report were obtained on two supercomputers, a mini-supercomputer, and a workstation installed at the Numerical Aerodynamics Simulation (NAS) Project at NASA Ames. These machines are described below.

The NAS CRAY Y-MP is an eight CPU, multiple instruction multiple data stream (MIMD), shared memory computer system from CRAY Research, Inc. (CRI). The clock period (CP) is 6.0 nanoseconds (*ns*), and there are 256 MW of main memory. The operating system (UNICOS 6.1) is UNIX System V based. Performance results described herein were obtained by running the Y-MP in dedicated time, although workload results closely reproduce the dedicated time results.

The NAS CRAY-2 is a four CPU, MIMD, shared memory computer system. The clock period is 4.1 *ns*, and there are 256 MW of main memory. The CRAY-2 operating system is UNICOS 6.1. Unlike the Y-MP, the CRAY-2 does not allow multiple and overlapping memory bank accesses. CRAY-2 workload results show poorer performance than do the dedicated time results reported here.

The NAS Convex C3240 is a four CPU, MIMD, shared memory computer system. The clock period is 40 *ns*, and there is one GByte of main memory. The ConvexOS operating system is partially derived from BSD UNIX release 4.2.

This study used a single-CPU Silicon Graphics, Inc. (SGI) 4D/25 workstation with 16 MBytes of RAM, a 32 KByte data cache, and a 64 KByte instruction cache. The processor was a 20 MHz MIPS R3000, with a R3010 floating point co-processor. The IRIX 4.0 operating system is based on System V Release 4.

Convex and SGI performance results were obtained using CPU times measured in lightly loaded environments. The shortest CPU time in a series of measurements was chosen as the best approximation to a dedicated time result.

1.2 Compiling Systems

The CRI FORTRAN compiling system, *cf77* [*7*], comprises a compiler (*cft77*), two pre-processors (*fpp* and *fmp*), and a loader (*segldr*). *Fpp* restructures code, inserts directives, and makes library substitutions in preparation for *fmp*, which parallelizes the code. *Cf77* is a standard FORTRAN 77 compiling system with CRI extensions. These extensions include compiler directives for vectorization and parallelization, and Fortran 90 array constructs. This study was performed with *cft77* version 5.0.1.18, *fpp* version 5.0, and *segldr* version 6.0.

The CRI C compiler, *scc* [*8*], incorporates the functions of both an Standard C pre-processor and a compiler. *Scc* invokes *segldr* to link the object files. The compiler accepts directives in the form of #pragma statements, and has CRI extensions to the standard, including variable-dimension arrays and complex arithmetic. This study was performed with *scc* release 3.0.3.3.

The Convex FORTRAN compiler, *fc* [*9*], is a single-step compiler which then invokes the link editor (*ld*) to link the object files. The compiler (version 7.0) supports numerous extensions, including vectorization and parallelization directives and Fortran 90 array constructs.

The Convex C compiler, *cc* [*10*], combines the pre-processor and compiler into a single executable. *Cc* version 4.3.2.0 was used in this work. Cc invokes *ld* to link the object files.

SGI supplies a FORTRAN compiler, *f77* [*11*], and a C compiler, *cc* [*12*]. Both compilers support the relevant standards as of IRIX release 4.0. The GNU project C compiler (*gcc*), which supports the C standard, was also used in this study. Versions 3.4.1, 2.0.1, and 2.0.2 of *f77*, *cc*, and *gcc*, respectively, were used in this study.

1.3 Data Storage

The basic floating point type in FORTRAN (REAL) and C (float) occupies 64 bits of memory on CRI machines. The floating point mantissa is represented by 48 bits in hardware, although only 47 bits are guaranteed by the model specified in the C documentation [*8*]. The basic integer type (INTEGER and int for FORTRAN and C, respectively) uses 46 bits by default. Compiler switches can be used to change this to 64 bits. Floating point instructions in the executable code produced by *scc* and *cf77* approximate floating point rounding in the same manner.

INTEGER and int words are 32 bits long on both the C3240 and the 4D/25. REAL and float are also 32 bits long on these machines. The results for these machines reported in this paper are obtained using DOUBLE PRECISION or double types for FORTRAN and C, respectively. These are 64 bit words in each case, 53 bits of which are assigned to the mantissa. The C3240 data were obtained with the IEEE floating point mode, which uses the IEEE format and does not use IEEE arithmetic [*13*].

2 Differences between C and FORTRAN

The original FORTRAN code was contained in a single file. The Standard C port was divided into eleven files. Each kernel is packaged together with its driver into a single file, for a total of seven files. (Both FFT kernels are contained in the same file.) The main routine and the timing routine each have their own file. There are two header files: one contains global constants and type definitions, while the other contains the complex arithmetic macros and typedef.

The major implementation details of this port are described below. A more detailed discussion of the issues pertaining to the use of C for numerical computations may be found in [*14*].

2.1 Derived types

2.1.1 Complex numbers

Unlike FORTRAN, C does not have a built-in type for complex numbers. As no direct translation is possible, the following policies were adopted for use in this port.

No vendor-specific extensions were used. In particular, CRI provides a complex data type which may be manipulated with the standard operators; this type was avoided. Macros were used as much as possible to substitute for the FORTRAN operators and intrinsic functions. Macros were written for sums, differences, products, ratios, inverses, exponentials, and assignment of complex numbers. Macros were also used for multiplication by real and imaginary constants. Macros (implemented with *#define* statements) were preferred over function calls for performance reasons, as they do not rely on a compiler's ability to inline functions.

While it was necessary to re-write many statements, the FORTRAN order of arithmetic operations was retained whenever possible. It was sometimes necessary to break up a single FORTRAN statement; in these cases, the alternate configuration requiring the fewest macro calls was chosen.

2.1.2 Floating point type definition

In order to facilitate porting between machines with differing floating point precision, a *real64* type was defined. This type is set to *float* in the main include file. Changing the typedef to *double* allows 64 bit arithmetic on most machines with 32 bit floating point.

2.2 Arrays

Automatic, statically allocated, arrays are declared with a syntax similar to FORTRAN arrays, to wit: float x[10][5]. The obvious syntax difference from FORTRAN is due to the fact that an *n*-dimensional C array is an array of pointers to (*n*-1)-dimensional arrays for *n* (*n*>1). In addition, the ordering of indices is reversed. The above example defines the same storage in memory as the FORTRAN statement: REAL X(5,10).

C arrays are addressed with zero-based indices. This differs from the FORTRAN default of one-based indices. C does not provide the flexibility which FORTRAN supplies with declarations such as: REAL X(5:10,10), in which arbitrary initial indices may be specified.

Standard C does not allow the user to pass FORTRAN-style variable-dimension multi-dimensional arrays. All dimensions except for the left-most (slowest-running) must be completely specified at compile time. This restriction is due to the fact that additional dimensions require new arrays of pointers. CRI's C compiler allows the user to pass FORTRAN-style arrays, but this extension was not used here.

The C standard allows programmers to work around some of the above problems. Variable-dimension multi-dimensional arrays may be emulated in a variety of ways using pointers and dynamic memory allocation. This issue is not addressed here, but the reader should note that the performance of a code will vary depending on whether dynamic or automatic arrays are used, and and the type of dynamic array which is implemented. There appears to be no general solution for the restriction to zero-based arrays. A common work-around [*15*] uses pointer arithmetic to adjust the origins of arrays outside of their original bounds. This relies on behavior which the C standard leaves undefined, and thus may not be used in a standard-conforming program [*1*].

2.3 Parameters

The FORTRAN parameter statement allows the programmer to associate a constant with a symbolic name inside a single program unit (subroutine, function, or main program). While C does not have an exact analogy, the *cpp #define* statement provides similar functionality. During the pre-processor phase of compilation, all occurrences of the symbolic name given in the *#define* statement are replaced with the constant specified by the statement. The compiler thus compiles code with a constant, achieving the same effect as the PARAMETER statement. By placing the *#define* statements inside include files, an advantage is gained over FORTRAN in that a constant is guaranteed to be the same in every routine which uses that include file. A practical disadvantage of using *#define* statements is the inability of many debuggers, including the CRI debugger, to examine the values of constants so defined.

Certain kernel routines had dummy arguments corresponding to PARAMETER constants in their driver routines. These arguments were deleted from the C kernel's formal parameter list, as the data was available through the #define and #include mechanisms.

C also provides a *const* qualifier which allows the programmer to specify in a variable's declaration that it is not to be changed. The intent of *const* may be evaded by using pointers. While the standard specifies that the behavior of such code is undefined, most compilers fail to detect this evasion unless particular flags are set. The *const* qualifier thus was not considered a reasonable replacement for the PARAMETER statement.

2.4 Commons

The NAS Kernels use a working space which consists of 360,000 words of named COMMON. This COMMON is included in all kernel drivers, and some kernels. In the remaining kernels, the data is passed in as dummy arguments. In the C port, the data arrays are defined in each driver. For kernels which accessed data *via* COMMON, the data arrays are given file scope with local linkage. Otherwise, the arrays are passed in the parameter list.

2.6 Function Prototypes

Standard C allows the programmer to write prototypes specifying the usage of each function before it is used. This allows the compiler to type-check each invocation of a function. All functions in this port were prototyped.

2.7 New Operators

The C programming language provides a number of assignment operators in addition to the "=" operator available in FORTRAN. These operators allow the programmer to combine arithmetic operations with assignments to a variable, decreasing the likelihood of such errors as typing mistakes. These operators include: +=, -=, *=, and /=, where for example, the statement "x=x+y" may be replaced by "x+=y".

2.8 Required User-written macros

Several commonly used FORTRAN intrinsic functions and operators do not have analogous standard macros or functions in C. Those which were relevant in this port were the *amin* and *amax* functions, and the ** operator. Three macros, named *max, min,* and *square,* were written to provide comparable functionality.

2.9 Order of Evaluation

FORTRAN allows the programmer to specify the order in which an expression is evaluated by the use of parentheses. Parentheses may be used to achieve the same end in C, although in more limited circumstances. In particular, using parentheses allows the programmer to specify an order of evaluation which *differs* from that obtained with the default precedence rules. When grouping operators of equal precedence, as in a sum of four variables, the compiler may choose any correct order of evaluation irrespective of parentheses present in the expression. A standard-conforming C compiler may not vary the order of evaluation.

2.10 NAS Kernels COPY Routine

The NAS Kernels COPY subroutine, which is used to copy saved data into a kernel's input data, was not ported. Rather, the Standard C library routine *memcpy* was used. This routine is vectorized on the CRI machines.

3 Automatic Code Translation

The use of *f2c* [16], an automatic FORTRAN-to-C code translator, was considered for this project, despite the fact that its authors suggest it is more appropriate for use as a compiler than as a translator. They warn that *f2c* produces C code which is cryptic, difficult to maintain, and bears little resemblance to the original FORTRAN source code. This warning proved to be well-founded, due in part to the fact that array computations are implemented using explicit pointer arithmetic. A brief investigation of the translated code found that the performance of the kernels containing complex arithmetic was limited by the fact that complex operations were implemented as functions rather than macros.

4 Results

Performance results for the un-modified kernels are given in Table 2 below. (Some compilers could not generate 64 bit floating point code for the kernels as written.) Table 3 contains the error as reported by each kernel. In each case, the compilation options which attained the best performance were used. The differences between the CRAY-2 and Y-MP errors are due to differences in floating point arithmetic. The Convex and SGI machines are expected to have better precision, due to longer mantissas (*cf.* sect. 1.3).

Table 4 shows performance results for those compilers where minimal line changes were required. Five lines were changed for the C code on the Convex. First, the *real64* type was re-defined as *double.* Also, local arrays in *fft* and *cholesky* were declared as static due to a compiler-imposed limit on the size of stack-allocated arrays. In porting the C code to the SGI, the only line change required was the *real64* type re-definition.

Table 2: Performance (in MFLOPS) with no line changes

Computer	Compiler	mxm	fft	Chol.	btrix	gmtry	emit	vpnta	Total
Y-MP	scc	250	74.8	88.8	140	131	183	50.1	102
	cf77	305	76.3	90.9	153	251	183	51.9	111
CRAY-2	scc	199	10.0	27.2	53.7	28.0	149	9.26	22.8
	cf77	417	11.3	27.1	58.1	176	148	9.23	24.2
C3240	fc	35.2	7.10	12.5	13.6	21.5	25.3	3.73	10.2

The FORTRAN kernels which did not use complex variables ran with good precision on the SGI with no line changes. However, 25 line changes were required to change the *cmplx, real,* and *aimag* intrinsics to *dcmplx, dreal,* and *dimag* in the remaining kernels. The latter three intrinsics are SGI extensions for their *DOUBLE COMPLEX* type, and are not part of FORTRAN 77. These changes violate the rules for official FORTRAN NAS Kernels performance measurements [5].

Table 3: Fractional error x 10^{13}

Computer	Com-piler	mxm	fft	Chol.	btrix	gmtry	emit	vpnta	Total
Y-MP	scc	0.517	32.00	1825.6	29.72	8.380	1.561	12.17	1910
	cf77	1.809	32.00	1825.6	60.62	6.561	1.561	2.354	1931
CRAY-2	scc	0.0646	2.104	768.8	64.35	115.2	0.473	8.265	959.2
	cf77	0.1292	2.104	768.8	74.89	97.81	0.473	7.313	951.5
C3240	cc	0.0303	1.289	65.97	5.724	2.240	0.132	0.0877	75.47
	fc	0.0303	1.289	65.97	5.724	0.867	0.108	0.0877	74.07
SGI 4D/25	cc	0.0283	0.374	28.78	0.0562	0.450	0.027	0.418	30.14
	gcc	0.0283	0.374	28.78	0.0562	0.450	0.027	0.418	30.14
	f77	0.0343	0.374	28.78	2.503	2.588	0.025	0.085	34.39

Table 4: Performance results with "minimal" line changes

Computer	Com-piler	mxm	fft	Chol.	btrix	gmtry	emit	vpnta	Total
C3240	scc	35.5	5.01	11.3	19.3	15.4	15.2	3.74	9.03
SGI 4D/25	cc	2.67	1.52	0.94	1.75	1.01	3.32	0.97	1.48
	gcc	2.27	1.00	0.83	1.56	0.87	2.76	0.92	1.22
	f77	3.06	1.50	0.91	1.75	1.05	3.23	1.04	1.52

4.1 C Compiler Options

A C compiler may, unlike a FORTRAN compiler, fail to vectorize loops due to potential aliasing of variables through pointers. A more successful analysis will result in maximum safe vector lengths, although these may be much smaller than the hardware vector size. Additionally, a C compiler must analyze possible loop recurrences and complex loop indices, as does a FORTRAN compiler.

The above analyses can be bypassed by supplying a command line option which instructs the compiler to insert vectorization directives before each loop. CRI's *-hivdep* option performs this function; Convex has no such option. This option is both effective and dangerous, as it allows the user to force the vectorization of loops for which recurrence warnings are correct.

The CRI *-hrestrict=f* option informs *scc* that all pointers are *restricted pointers*. A restricted pointer is assumed to point to objects which are not referenced through any other pointers. This allows the compiler to optimize code assuming that no aliasing occurs. As in FORTRAN, the programmer must ensure that the assumption is correct.

Convex *cc* may be invoked in a standard-compliant mode, or with language extensions. *Cc* accepts command line options (*-alias array_args*, *-alias ptr_args*) which together provide the same functionality as CRI's *-hrestrict=f*. The command line option which instructs *cc* to use 64-bit floating point fails to accept very large or very small constants; in these cases, the user must explicitly use the *double* floating point type.

The CRI *-hrestrict=f* and Convex *-alias array_args* and *-alias ptr_args* options will be referred to as *restricted pointer* options in the following.

The CRI *-hvector3* option initiates aggressive vector optimization. This includes expression re-ordering, partial vectorization for some loops with recursion, and additional dependence and alias analysis.

Both C compilers used on the 4D/25 provided options for increasing levels of scalar optimizations. Neither compiler allowed the automatic promotion of *float* to *double*.

4.1.1 CRI

Vpenta was the only kernel to achieve its best performance without invoking compiler options. All loops vectorized, and aggressive vectorization did not change the performance. The performance of the three kernels (*mxm*, *fft*, *cholesky*) which passed data arrays through the parameter list improved when the restricted pointer option was used. Additional gains were obtained for *fft* and *cholesky* on the Y-MP when aggressive vectorization was invoked, although the best *fft* performance was still obtained with *-hivdep*. Aggressive vectorization had no effect on CRAY-2 performance.

Table 5 shows the best performance attained without using -*hivdep*. Thus, automatic insertion of compiler directives can be avoided while incurring only a 5% performance penalty on the Y-MP. This penalty increases to 26% on the CRAY-2. Care must be taken in selecting options, however. It was found that using restricted pointers hurt the performance of kernels accessing global data on the Y-MP. There was no such penalty on the CRAY-2.

Table 5: CRI code performance without forcing vectorization directives

Kernel	Options: Y-MP	CRAY-2	Performance (MFLOPS): Y-MP	CRAY-2
mxm	*restrict=f*	*restrict=f*	250	199
fft	*restrict=f,vector3*	*restrict=f*	73.0	9.23
cholesky	*restrict=f,vector3*	*restrict=f*	88.8	25.5
btrix	*vector3*	none	124	18.8
gmtry	*vector3*	none	96.2	22.1
emit	*vector3*	none	182	146
vpenta	none	none	50.1	9.26
Total	—	—	97.2	17.9

The superior performance of *mxm* and *cholesky* seen with restricted pointers may be due to the fact that when *-hivdep* is used, *scc* may still issue CMR (complete memory references) instructions if it finds possible aliasing [8]. This degrades performance by forcing otherwise asynchronous memory loads to wait until stores are complete. When restricted pointers are used, the compiler will not issue CMRs.

Table 6: CRI Loops vectorized without using *-hivdep* compiler option

Loops:	Y-MP 8/256			CRAY-2 4/256		
Kernel	Vector-ized	Max. Safe Vect. Len.	Vector-izable	Vector-ized	Max. Safe Vect. Len.	Vector-izable
mxm	2	0	2	2	0	2
fft	4	0	4	4	0	4
cholesky	9	0	9	9	0	9
btrix	7	0	7	7	0	7
gmtry	4	1	6	4	1	6
emit	6	1	8	6	1	8
vpenta	7	0	7	7	0	7
Total	39	2	43	39	2	43

The *-hstdc* option,which imposes strict compliance to the C standard, reduced the flop rate of *cholesky, gmtry,* and *emit.* The performance changed because the standard requires range-checking for certain mathematical functions; the versions of these functions which implement range-checking do not vectorize.

4.1.2 Convex

All kernels were compiled with the *-O2* option, which invokes vectorization and function-wide scalar optimization. In addition, the restricted pointer option was used for the kernels which accessed data *via* the parameter list. The compiler was able to completely vectorize five of the seven kernels (*cf.* Table 7). *Gmtry* and *emit* were only partially vectorized. All loops which failed to vectorize contained complex arithmetic macros; the compiler reported finding "no induction variable" in these cases. Other loops containing the same macros succeeded in vectorizing, however. As on the CRI machines, the performance of *cholesky, gmtry,* and *emit* degraded when the standard-enforcing compiler options (*std* and *str*) were used. These results, which used the IEEE floating point mode, were checked against runs which used the native floating point mode. No differences were found between the two modes.

Table 7: Number of Convex Loops vectorized

Kernel:	Vectorized	Vectorizable
mxm	2	2
fft	4	4
cholesky	9	9
btrix	7	7
gmtry	1	6
emit	4	8
vpenta	7	7
Total	34	43

4.1.3 SGI

All kernels performed best using *cc -O2 cc* option on the SGI. The *-O3* option, which added global register optimization, degraded performance by up to five percent. The *-cord* option, which rearranges procedures to improve caching and paging, had no effect. The best performance using *gcc* on the SGI was obtained with the *-O2* option, which performs register and loop optimizations.

4.2 FORTRAN Compiler Options

CRI executables discussed in this report were compiled using *cf77* -Zv naskern.f, corresponding to processing a code with *cft77, fpp,* and *segldr*. Convex executables were obtained with *fc* -cfc naskern.f -lm -lcfc. SGI executables were compiled with *f77 -r8* naskern.f -lm.

4.3 Library substitutions

The FORTRAN versions of *mxm* and *gmtry* ran significantly faster than did the C versions on both CRI machines. *Fpp* substituted library calls for source code in both of these kernels. The matrix multiply nested loop was eliminated in *mxm,* and a rank-one update was replaced at the end of *gmtry*. *Scc* did not recognize the analogous C code as candidates for library substitutions. Runs comparing the performance of the FORTRAN code with and without the substitutions showed that the *mxm* performance gain due to this effect was 9% and 90% on the Y-MP and the CRAY-2, respectively. The improvement for *gmtry* was even more dramatic, being 120% and 530%, respectively.

4.4 Effect of assignment operators

As noted in section 2.7, C provides several assignment operators which are not available in FORTRAN. Surprisingly, the performance and precision of two kernels on the CRI machines were affected by the use of these operators. The error reported by *mxm, btrix* and *vpenta* changed to agree with the errors reported by FORTRAN code when the "=" operator was used on the Y-MP. More importantly, the performance of *mxm* improved to 280 MFLOPS with the use of the "=" operator. The performance of *btrix* changed only slightly, increasing to 143 MFLOPS; *vpenta*'s performance increased to 51.9 MFLOPS.

Making the same change on the CRAY-2 resulted in a small improvement in the *mxm* flop rate (to 200 MFLOPS), a small decrease in the *btrix* flop rate (to 52.8 MFLOPS), and no change in the flop rates of the other kernels. As on the Y-MP, the errors reverted to the FORTRAN values.

Further studies showed that this effect was due to the order of evaluation imposed by the "+=" operator, in which the right hand side of an expression is evaluated *before* it is added to the left hand side. Using the "=" operator resulted in all variables being summed simultaneously. An important improvement on the Y-MP would be to allow the "+=" operator to be evaluated with performance comparable to the "=", either automatically or through a compiler option which the user could use to allow re-arrangement of expression evaluation.

While the Convex C compiler was not sensitive to this effect, both *gcc* and *cc* showed the opposite behavior on the SGI. The errors reported by all three kernels mentioned above were affected by the choice of assignment operators. Only *mxm* showed more than a minor dependence of the flop rate on operator. Using the "=" operator, *mxm*'s performance decreased to 2.18 and 2.57 MFLOPS with *gcc* and *cc*, respectively.

4.5 Explicit array indexing

A test was performed in which the arrays were accessed as one-dimensional arrays with indices derived from the multi-dimensional indices. The performance did not improve on the CRI machines, indicating that the intermediate pointers were not being used. The same results were observed when the test was performed for *mxm* using *cc* on the C3240, and both *cc* and *gcc* on the 4D/25.

4.6 Translation of Complex operations

As noted above (*cf.* Sect. 2.1.1), complex arithmetic was implemented using macros. The only kernel in which the order of evaluation was difficult to reproduce was *gmtry*. This was due to the appearance in two places of code of the form:

Z1 = EXP((W1 - W2) * P)

Here, Z1, W1, and W2 are complex, while P is real. The error obtained by ordering the C statements as difference, multiply, and exponential (Order 1) best reproduced the error found on the Y-MP and CRAY-2 (*cf.* Table 3). Calculating the exponentials, and then dividing (Order 2) decreased the error by a factor of five on the CRAY-2, while increasing it by a factor of eight on the Y-MP.

Both of the above orders of evaluation produced the same error on the C3240. Order 2 achieved the best agreement (< 1%) between the C and FORTRAN errors on the SGI for *cc* and *gcc*. The choice of order of evaluation did not affect the performance of *gmtry* on any of the tested machines.

5 Precision of results

5.1 Dependence of precision on compiler options

On all tested machines except the SGI, the errors reported by some kernels changed when certain compiler options were tried. The relevant options are the *-hstdc* and *-hvector3* options on the CRI computers, and the *-std*, and *-str* options on the Convex computer. In no case was the resulting error unacceptably high.

As noted above (*cf.* Sect. 4.1), the *-hstdc* option inhibited the vectorization of four kernels. The errors of *gmtry* and *emit* changed on both the CRAY-2 and the Y-MP, while that of *fft* changed on the CRAY-2 alone. This change appears to be due to the fact that different versions of the mathematical functions are called with the *-hstdc* option, rather than the lack of vectorization. Runs of un-vectorized executables compiled without the *-hstdc* option reported the same errors as are shown in Table 3. The *-hvector3* option resulted in changed errors for *btrix* and *vpenta* on the Y-MP. This option causes aggressive optimization, and small changes in the error may not be surprising.

The errors reported by all kernels except *cholesky* and *gmtry* changed when either the *-std* or *-str* options were invoked on the C3240. These options progressively more restrictive adherence to the standard. As on the CRI machines, the compiler uses different versions of the mathematical routines with these options. This may account for the change in the error.

5.2 Tests of reference answers

The reference answers were originally obtained by running the kernels on a VAX using extended floating point. As the error reported by the C and FORTRAN NAS Kernels differed on all machines tested, several tests were performed to determine whether smaller reported errors represented more precise answers. The C and FORTRAN codes were run with 128 bit floating point arithmetic on the Y-MP using the *long double* and *DOUBLE PRECISION* data types, respectively. In addition, a multiple-precision arithmetic package [*17*], developed by D.H. Bailey at NAS, was used to generate answers using 256 bit floating point arithmetic for all kernels except *gmtry*, and *emit*. (The multiple precision versions of the latter two kernels were not tested due to the prohibitive amount of CPU time required.) The results of these tests are summarized in Table 8.

Table 8: Fractional deviation of reported Y-MP result from reference value $\times 10^{13}$

	mxm	fft	Cholesky	btrix	gmtry	emit	vpenta
128 bit FORTRAN	.0028	0.73	41	3.5	5.3	0.17	.0056
128 bit C	.0028	1.3	41	3.5	0.66	0.17	.0056
256 bit MPFUN [*16*]	.0028	1.2	41	17	—	—	.0056

Comparing the results in Table 8 to the errors reported in Table 3, it appears that the smaller errors reported for *mxm*, *cholesky*, and *vpenta* reflect more precise answers. Further, all differences between 64 bit CRI results and the reference values exceed those given in Table 8, indicating that the smaller errors do result from more precise results on the Y-MP and the CRAY-2. Detailed investigation beyond the scope of this report would be required to assess the relative precision of the SGI and Convex results, although they are both clearly more precise than the CRI calculations. The effect of CRI floating point arithmetic on Cholesky factorization, which has the largest fractional deviation in Table 8, has been studied in detail[*18*].

6 Remarks

Several issues have been raised by this study. While C code written using automatic arrays performs well, many programmers will require (at least) the flexibility offered by FORTRAN variable-size arrays. To this end, data on the performance of various strategies for dynamically allocated arrays is needed. Programmers may also be interested in using structs to define their own data structures. The performance of code using structs remains to be studied.

In addition to the language and performance issues just mentioned, there are several respects in which the C compilers studied need to be improved. None of the C compilers studied produced full source code listings with diagnostics. At best, optimization messages were printed with line number references which did not always correspond to the line numbers used by text editors.

The CRI and Convex C compilers, unlike the SGI C compilers and the various FORTRAN compilers, required additional command line options beyond such simple switches as *-O2*. While this is appropriate for such actions as automatic insertion of compiler directives, it may be best to incorporate optimizations based on restricted pointers as the defaults for aggressive vectorization levels (*i.e. -hvector3* and *-O2* for CRI and Convex, respectively). Further, some of the analysis and optimization presently performed with the CRI *-hvector3* option seems to be more appropriate to the (default) *-hvector2* option, as it corresponds to optimizations performed by *cf77* with no options.

The Convex vectorization analysis, which was successful in most kernels, had problems with some loops involving complex macros in *gmtry* and *emit*. The CRI vectorization analysis also needs some improvement, as evidenced by the superior performance of most kernels compiled with the *-hivdep* option.

The performance results reported here do not necessarily reflect the best results which a programer can achieve when tuning a code for performance. Rather, they correspond as nearly as possible to NASKERN tuning level 0 [5]. Similar changes to the C and FORTRAN codes may yield different performance gains.

Variation of code performance with problem size was not addressed in this study. Changes in the sizes of various arrays in these kernels may affect the FORTRAN and C versions differently.

6.1 Problems

Several problems should be noted here. Performance on the Y-MP drops when the restricted pointer option is invoked. As **no** change is performance is expected, this effect is unwelcome. When the += operator is used to increment array elements, *mxm* suffers a 11 % performance loss on the Y-MP, while *vpenta*'s performance drops by 4 %. A remedy for this problem should be part of the default compiler optimizations.

The Convex C compiler imposed a limit, not found in other compilers, on the size of automatically allocated arrays. The automatic conversion to 64 bit floating point failed to correctly handle very large constants.

Both C compilers on the 4D/25 showed a *mxm* performance drop of 5% when the = operator was used. Again, default code optimization should be insensitive to the choice of assignment operator.

7 Summary

The NAS Kernels have been ported to Standard C. Performance comparable to the original FORTRAN code was obtained for all platforms tested, although in each case command line arguments were required. Errors were within the bounds specified for the original benchmark. The principal difficulty in re-writing the code was changing the array indices.

The *f2c* program was also used to generate C code. This code was difficult to maintain and modify. The performance was poor, principally due to the use of functions rather than macros for complex arithmetic. For small codes such as the NAS Kernels, a hand-coded port is clearly the best approach. For large codes, it may be worthwhile to investigate better automated tools.

Acknowledgements

I would like to thank David Browning and Russell Carter of NAS RND for many useful discussions. I would also like to thank Bob Ciotti of NAS RND, who provided the original impetus for this work.

References

[1] American National Standard Programming Language C, ANSI X3.159 - 1989. American National Standards Institute, 1430 Broadway, New York, NY 10018.

International Standard Programming Language C, ISO/IEC 9899: 1990 (E). ISO/IEC Copyright Office, Case Postale 56, CH-1211 Geneve 20, Switzerland.

[2] American National Standard Programming Language FORTRAN, ANSI X3.9 - 1978. American National Standards Institute, 1430 Broadway, New York, NY 10018.

[3] American National Standard Programming Language FORTRAN, ANSI X3.198 - 1992. American National Standards Institute, 1430 Broadway, New York, NY 10018.

International Standard Programming Language Fortran, ISO/IEC 1539: 1991 (E). ISO/IEC Copyright Office, Case Postale 56, CH-1211 Geneve 20, Switzerland.

[4] D. H. Bailey and John T. Barton, "The NAS Kernel Benchmark Program," NASA Technical Memorandum 86711, NASA Ames Research Center, Moffett Field, CA 94035.

[5] D. Browning, "NAS Kernels Survey Report," Report RND-92-003, February 1992, NAS Systems Division, NASA Ames Research Center, Moffett Field, CA 94035.

D. H. Bailey, "NAS Kernel Benchmark Results," *First International Conference on Supercomputing Systems,* IEEE Computer Society, 1985, pp. 341-5.

[6] R. Carter, "NAS Kernels on the Connection Machine," Report RND-90-005, February 1991, NAS Systems Division, NASA Ames Research Center, Moffett Field, CA 94035.

[7] CRAY Research Inc., *CF77 Compiling System, Vol. 1*, Pub. No. SR-3071 5.0, 1991; *ibid., Vol. 4*, Pub. No. SG-3074 5.0, 1991.

[8] CRAY Research Inc., *Cray Standard C Programmer's Reference Manual,* Pub. No. SR-2074 3.0, 1991.

CRAY Research Inc., *Volume 2: UNICOS C Library Reference Manual,* Pub. No. SR-2080 6.0, 1991.

[9] Convex Computer Corp., *Convex Fortran Guide,* Order No. DSW-038, 1991.

[10] Convex Computer Corp., *Convex C Guide,* Order No. DSW-086, 1991.

[11] Silicon Graphics, Inc., *Fortran 77 Programmer's Guide,* Document No. 007-0711-030, 1991.

Silicon Graphics, Inc., *Fortran 77 Language Reference Manual,* Document No. 007-0710-040, 1991.

[12] Silicon Graphics, Inc., *C Language Reference Manual,* Document No. 007-0701-040, 1991.

[13] IEEE Standard for Binary Floating Point Numbers, ANSI/IEEE Standard 754-1985, IEEE, New York, 1985.

[14] T. MacDonald, "C for Numerical Computing," *The Journal of Supercomputing*, **5** (1991) 49-55.

[15] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vettering, *Numerical Recipes in C,* Cambridge University Press, Cambridge, U.K. 1988.

[16] S.I. Feldman *et al.*, "A Fortran-to-C Converter," Computing Science Technical Report No. 149, October 1991, AT&T Bell Laboratories, Murray Hill, NJ 07974.

[17] D. H. Bailey, "A Portable High Performance Multiprecision Package," Report RNR-92-022, June 1992, NAS Systems Division, NASA Ames Research Center, Moffett Field, CA, 94035.

D. H. Bailey, "Automatic Translation of Fortran Programs to Multiprecision," Report RNR-92-025, April 1992, NAS Systems Division, NASA Ames Research Center, Moffett Field, CA, 94035.

[18] R. Carter, "Y-MP Floating Point and Cholesky Factorization," *Int. J. High Speed Computing*, **3** (1991) 215-222.

NAS

RND TECHNICAL REPORT

Title:

"A Standard C Port of the NAS Kernels Benchmark Program"

Author(s):

I. E. Stockdale

Reviewers:

"I have carefully and thoroughly reviewed this technical report. I have worked with the author(s) to ensure clarity of presentation and technical accuracy. I take personal responsibility for the quality of this document."

Two reviewers must sign.

Signed: Alfred E. Nothaft

Name: Alfred E. Nothaft

Signed: [illegible]

Name: Russell C. Carter

After approval, assign RND TR number.

Branch Chief:

Approved: [illegible]

Date:

TR Number:

Important: Put this form as the last page in the published Tech Report.